Trabajo Práctico Final

Integrantes:

Lucas Alvarez - mail: alvarezlucas2787@gmail.com

Sciacca, Lucia - mail: luly_sciacca@hotmail.com

Martin, Nicolas - mail: niko.gm11@gmail.com

Decisiones del diseño:

Diseñamos al SEM como sistema de comunicación global entre diferentes objetos, que le informaran al mismo los registros de todos los movimientos realizados dentro de la aplicación.

Detalles de implementación:

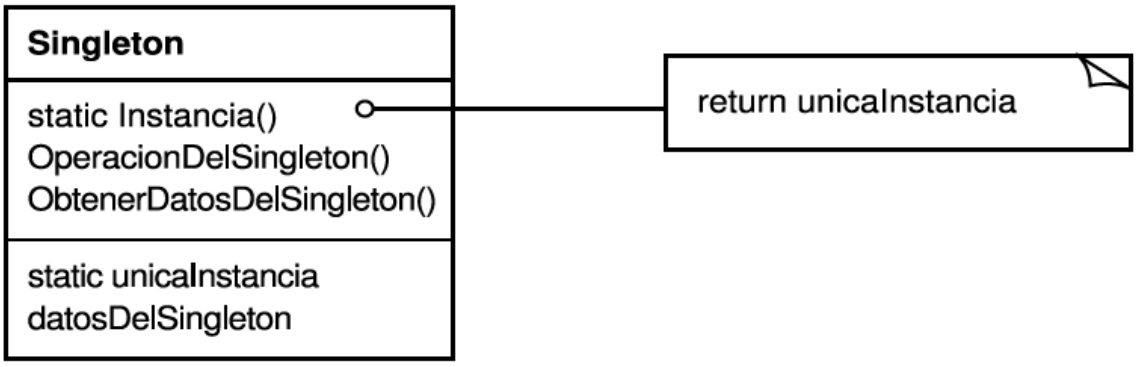
Se decidió que los estacionamientos sean ellos mismos quienes avisen al sem cuando estos hayan acabado. Luego se decidió que sean las zonas de estacionamiento quienes registren los estacionamientos y no el SEM, el SEM puede llegar a estos a través de las zonas. También, en el caso del Comercio, se deliberó que este le comunique al SEM cuando debe realizar una recarga a cierto número telefónico.

Patrones de diseño utilizados:

Patron Singleton:

Propósito:

Garantiza que una clase sólo tenga una instancia y proporciona un punto de acceso global a ella.



Participantes:

- Define una operación Instancia que permite que los clientes accedan a su única instancia.
- Puede ser responsable de crear su única instancia

Colaboradores:

Los clientes acceden a la instancia de un singleton exclusivamente a través de la operación Instancia de éste

Para el diseño del SEM se decidió implementarlo con el patrón de diseño Singleton, es decir que el sem solo podrá ser instanciado una vez y a la clase le es pedida esta instancia.

Patron State:

Propósito:

Permite que un objeto modifique su comportamiento cada vez que cambie su estado interno. Parecerá que cambia la clase del objeto.

Aplicabilidad:

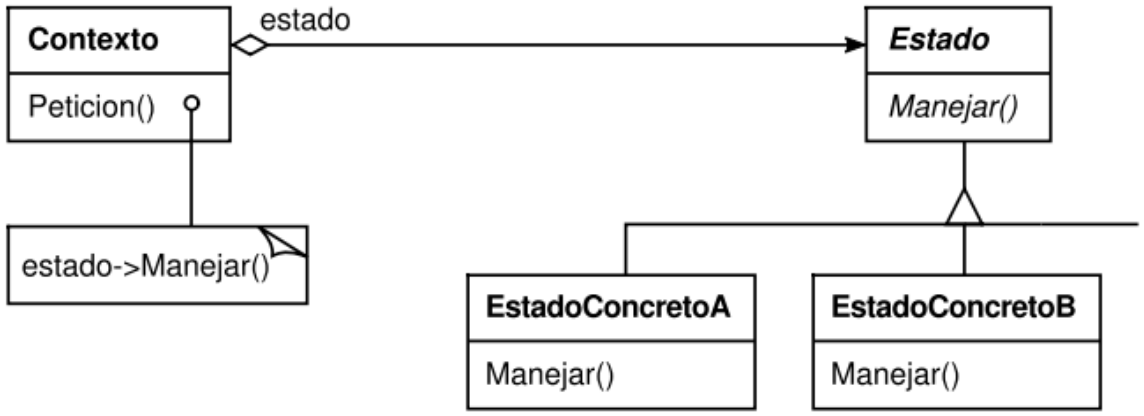
El patrón State se puede utilizar cuando:

- El comportamiento de un objeto depende de su estado y debe cambiar en tiempo de ejecución dependiendo de ese estado.
- Las operaciones tienen largas sentencias condicionales con múltiples ramas que dependen del estado del objeto. Este estado se suele representar por una o más constantes enumeradas. El patrón State pone a cada rama de la condición en una clase aparte. Esto nos permite tratar al estado del objeto como un objeto de pleno derecho que puede variar independientemente de otros objetos.

Pros y contras:

- Principio de responsabilidad única. Organiza el código relacionado con estados particulares en clases separadas.
- Principio open / closed. Introduce nuevos estados sin cambiar de estado existente o la clase contexto.
- Simplifica el código del contexto eliminando voluminosos condicionales de máquina de estados.
- Contra: Aplicar el patrón puede resultar excesivo si una máquina de estados tiene unos pocos estados o raramente cambia.

Estructura:



Se diseñó a los estados de la aplicación móvil con el patrón State, la app cuenta como un atributo que implementa la interfaz EstadoDeMovimiento que se irá intercambiando entre instancias de EstadoCaminando y EstadoConduciendo cuando a la app lleguen los mensaje walking() y driving() que implementa de la interfaz movementSensor().

Patrón Strategy:

Propósito:

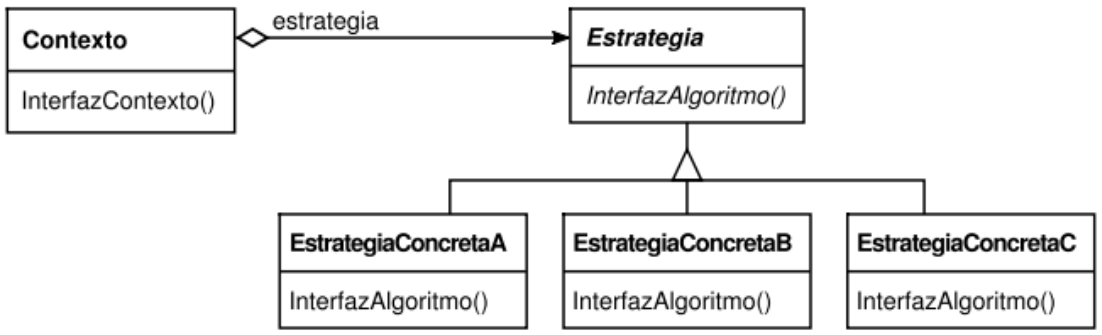
Define una familia de algoritmos, encapsula cada uno de ellos y los hace intercambiables. Permite que un algoritmo varíe independientemente de los clientes que lo usan.

Pros y Contras:

- Se puede intercambiar algoritmos usados dentro de un objeto durante el tiempo de ejecución
- Se puede aislar los detalles de implementación de un algoritmo del código que lo utiliza.
- Se puede sustituir la herencia por composición.

- Principio de open /closed. Puedes introducir nuevas estrategias sin tener que cambiar el contexto.
- Contra: Los clientes deben conocer las diferencias entre estrategias para poder seleccionar la adecuada.

ESTRUCTURA

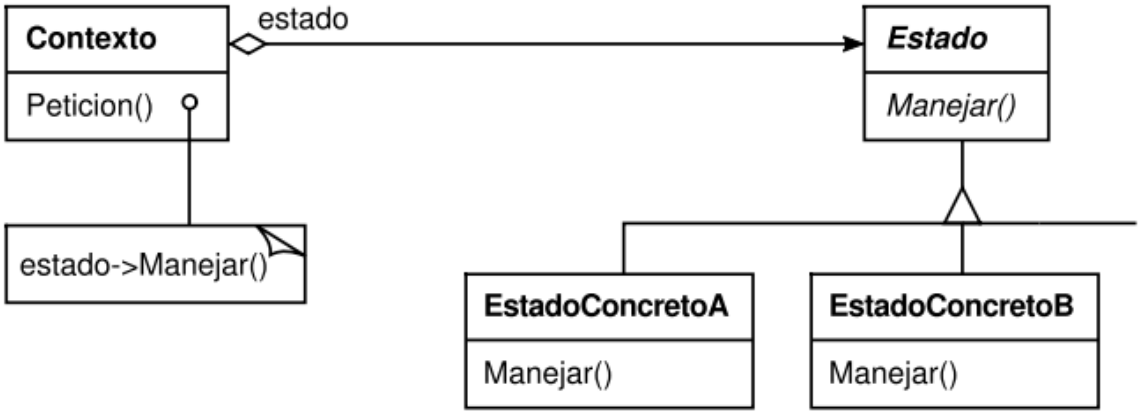


Para implementar los diferentes modos que tiene la app se decidio usar el patrón Strategy, la app, siendo el cliente, contará con una clase, ya sea Manual() o Automático() que implementan la interfaz ModoDeApp().Los estados de la app informan al modo de los diferentes cambios del estado del conductor y son los ModoDeApp quienes deciden los métodos que realizará la aplicación ya sea mostrar alertas o iniciar y finalizar los estacionamientos.

Patrón Observer:

Propósito:
Este patrón define una dependencia de uno a muchos entre objetos, de forma que cuando un objeto cambie de estado se notifique y se actualicen automáticamente todos los objetos que dependen de él.

Estructura:



Aplicabilidad:

Se utiliza este patrón cuando :

- Una abstracción tiene dos aspectos y uno depende del otro.Encapsular estos aspectos en objetos separados permite modificarlos y reutilizarlos de forma independiente.
- Un cambio en un objeto requiere cambiar otros, y no sabemos cuántos objetos necesitan cambiarse
- Un objeto debería ser capaz de notificar a otros sin hacer suposiciones sobre quienes son dichos objetos, o sea, cuando no queremos que estos objetos estén fuertemente acoplados.

Pros y contras:

- Principio de open/closed: Se pueden introducir nuevas clases suscriptoras sin tener que cambiar el código de la notificadora (y viceversa si hay una interfaz notificadora)
- Se pueden establecer relaciones entre objetos durante el tiempo de ejecución.
- Contra: Los suscriptores son informados de forma aleatoria.

En nuestro proyecto el SEM es nuestra clase notificadora que va a tener una lista de observadores, los que al implementar la interfaz ServicioDeAlerta podrán ser notificados de:

- El inicio de un estacionamiento
- La finalización de un estacionamiento
- La compra de recarga virtual